Binding python to other languages (Fortran and C)

Overview

- One of the beauties of python is the ease with which you can bind it to low-level programming languages.
- Allows python to be a scripting interface on top of optimised CPU-intensive processing code.
- Examples are CDAT and MetPy developed by ECMWF.
- Numerous packages are available to do this.
- Here we present Pyfort, F2PY for Fortran bindings and a quick look at C bindings.

Python/Fortran bindings

- For Fortran scientific Fortran programmers the progression to a new package involves:
    1. Learning of a new package/language
    2. Transferral of old code, re-writing, optimisation etc.
- These are barriers to switching.
- Imagine if you could just plug in your old functions and subroutines directly to your new package.
- Enter Python/CDAT, in association with **Pyfort** or **F2PY**.

Locating and installing the packages

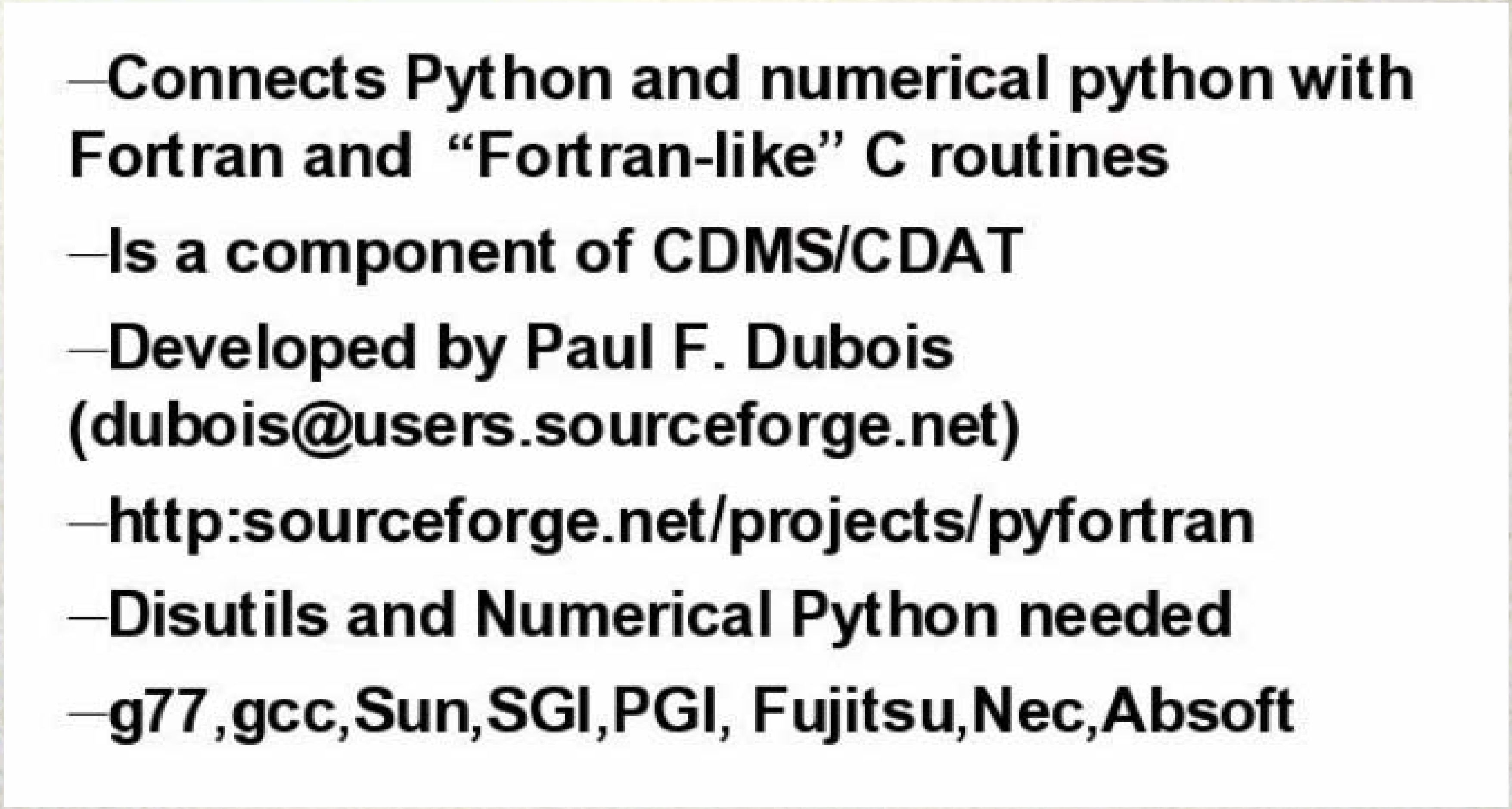
- You can freely **download** the packages at:
  - **Pyfort** - http://pyfortran.sourceforge.net
  - **F2PY** - http://cens.ioc.ee/projects/f2py2e

- **Installation:**
  - Both **Pyfort** and **F2PY** are now installed as part of CDAT and so is already available on a number of our linux machines under the directory:

```
/<your_cdat>/bin/[pyfort|f2py]
```

*Much of the information in this document was stolen from: http://www.prism.enes.org/WPs/WP4a/Slides/pyfort/pyfort.html

Pyfort Introduction



- –Connects Python and numerical python with Fortran and “Fortran-like” C routines
- –Is a component of CDMS/CDAT
- –Developed by Paul F. Dubois (dubois@users.sourceforge.net)
- –http:sourceforge.net/projects/pyfortran
- –Disutils and Numerical Python needed
- –g77,gcc,Sun,SGI,PGI, Fujitsu,Nec,Absoft

*Much of the information in this document was stolen from:
http://www.prism.enes.org/WPs/WP4a/Slides/pyfort/pyfort.html

Pyfort Usage: Overview (1)

- The interface to pyfort is relatively simple:

1. Pyfort takes a file or number of files holding Fortran functions and/or subroutines.
2. These are compiled and linked to a library.
3. The user then hand edits a Pyfort (**.pyf**) text file describing the interface to each function/subroutine.
4. The **pyfort** command is then run with the necessary arguments to produce some C code to describe the Fortran interface to python. Pyfort automatically compiles this C code into what is called a Python Extension Module (**.so**).
5. The Python Extension Module can then be imported directly into python with the functions/subroutines visible as module level python functions.

Pyfort Usage: Overview (2)

- This means that once you have created a Python Extension Module using Pyfort you will always have access to it at the Python level and, from the user’s perspective, it appears just like any other Python function.

Pyfort: A simple example (1)

- Below is a basic Fortran subroutine that has been connected to python. It demonstrates the use of the Pyfort interface without any complex code to confuse you:

- The **itimes.f** contains the subroutine **itimes** which takes in two Numeric arrays (x and y) of length n and returns an array (w) of the same length where w(i)=x(i)*y(i).

```
          subroutine itimes(x,y,n,w)
          integer x(*)
          integer y(*)
          integer w(*)
          integer n
          integer i
          do 100 i=1,n
            w(i) = x(i) * y(i)
   100    continue
          return
          end
```

Pyfort: A simple example (2)

The two subroutines where placed in the files '**addone.f**' and '**minusone.f**' and compiled them as follows:

```
g77 -c itimes.f
```

The compiled subroutines were then linked into a fortran library called **libitimes.a**:

```
ld -r -o libitimes.a itimes.o
```

Pyfort: A simple example (3)

You then need to write a Pyfort script declaring the parameters involved called **testpyf.pyf**:

```
SUBROUTINE ITIMES(X, Y, N, W)
! times (x,y,n,x) sets (i)=x(i)*y(i), i=1,n
integer, intent(in):: x(n), y(n) ! must have
                                   size n
integer, intent(out)::w(n)
integer n
END SUBROUTINE itimes
```

- Finally, run Pyfort with the following arguments to produce the C code that glues it all together (this allows you to call the module and functions from python):

```
pyfort -c g77 -i -l./itimes testpyf.pyf
```

Pyfort: A simple example (4)

- The output of this compilation was the production of a Python Extension Module called **testpyf.so** located at:

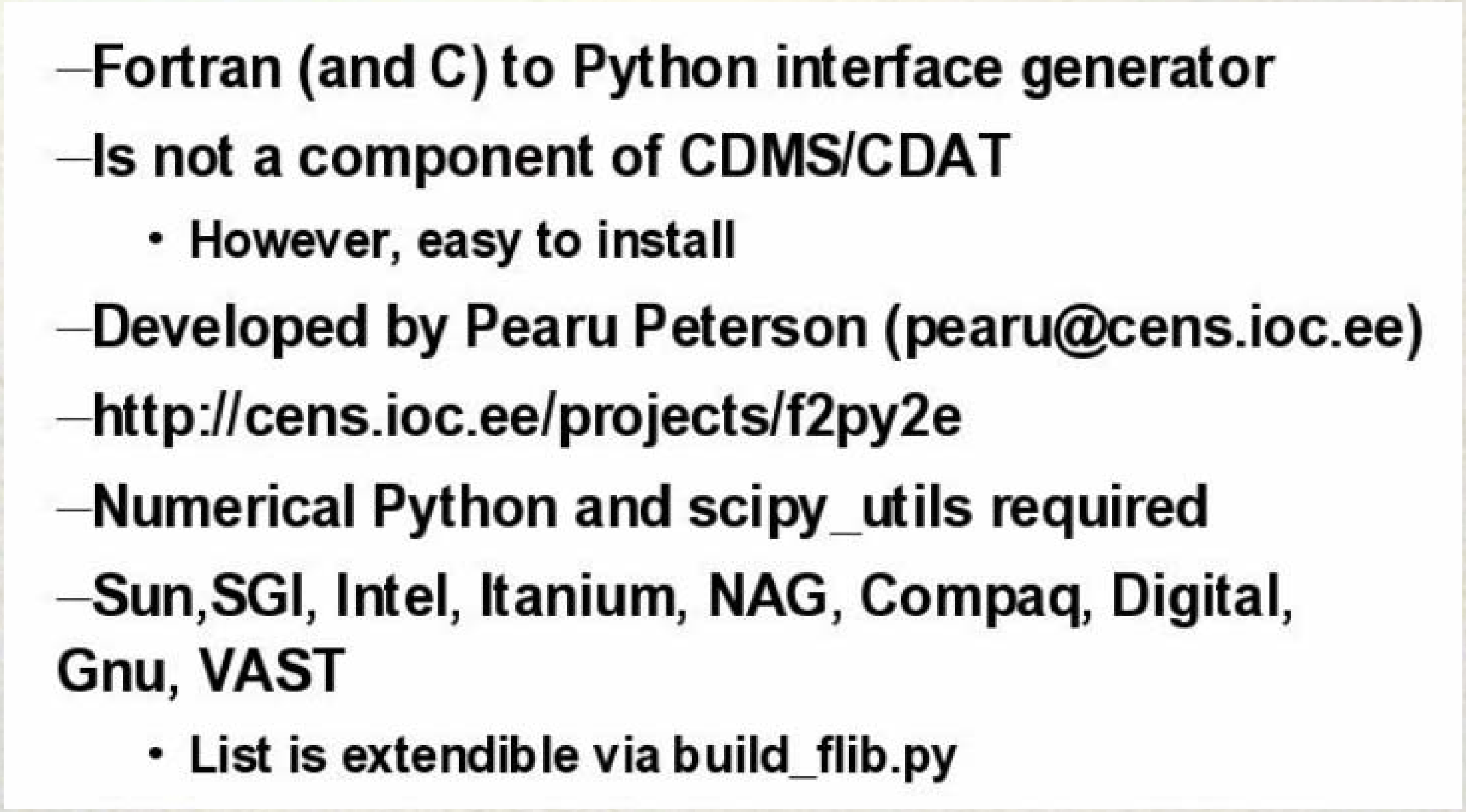
  ```
  build/lib.linux-i686-2.2/testpyf.so
  ```

- You can then import this module directly into python and call both subroutines as python functions:

  ```
  > import sys ;
     sys.path.append('build/lib.linux-i686-2.2')
  > import testpyf, Numeric
  > x=Numeric.array([1,2,3]) ;
     y=Numeric.array([4,5,6])
  > n=len(x) ; print "itimes", x, y
  itimes [1,2,3] [4,5,6]
  > print testpyf.itimes(x,y,n)
  [4,10,18]
  ```

F2PY Introduction

- You can freely download the packages at:



- Fortran (and C) to Python interface generator
- Is not a component of CDMS/CDAT
  - However, easy to install
- Developed by Pearu Peterson (pearu@cens.ioc.ee)
- http://cens.ioc.ee/projects/f2py2e
- Numerical Python and scipy_utils required
- Sun,SGI, Intel, Itanium, NAG, Compaq, Digital, Gnu, VAST
  - List is extendible via build_flib.py

*Much of the information in this document was stolen from: http://www.prism.enes.org/WPs/WP4a/Slides/pyfort/pyfort.html

F2PY Usage: Overview (1)

- F2PY demonstrates greater functionality than Pyfort, for example you can return character arrays, deal with **allocatable arrays** and **common blocks**, which pyfort does not allow.

- The F2PY interface is potentially simpler than that for Pyfort, but there are various methods you can choose from. The F2PY documentation takes you through these methods.

F2PY Usage: Overview (2)

- The following example below shows the simplest method where you can do everything in one line. Note that if you have arguments with the **intent ‘out’** or **‘inout’** then you will probably need to hand edit the **‘.pyf’** file or the original Fortran code.

F2PY: A simple example (1)

1. Create a fortran file such as **hello.f**:

```
C File hello.f
      subroutine foo (a)
      integer a
      print*, "Hello from Fortran!"
      print*, "a=",a
      end
```

2. Run F2PY on the file:

```
f2py -c -m hello hello
```

F2PY: A simple example (2)

- Run python and import the module, then call the subroutine as a function:

```
$ python
>   import hello
>   hello.foo(34)
‘Hello from Fortran!’
a= 34
```

Choosing between Pyfort and F2PY

- **F2PY** is the more comprehensive of the two packages (**providing support for returning character arrays, simple F90 modules, common blocks, callbacks and allocatable arrays**) but if pyfort does what you want, it may be easier to get to grips with.

- Both **Pyfort and F2PY are useful tools** and deciding on which one to use will depend on a number of issues. In theory, using either tool should be a quick (less than 1 hour) job but determining the duration will depend on issues such as:

How to choose

- Which package am I experienced with?
- Which package is available already on my platform?
- How long does it take to install (if not already present)?
- Which Fortran compiler am I using?
- Can I get away with the quick F2PY solution that involves no hand editing of files?
- Do I need to return character arrays from my subroutine (in which case you need to use F2PY)?
- Am I using callbacks (need F2PY again)?
- Do I need to handle F90 modules (need F2PY again)?
- Do I need to use Common Blocks (need F2PY again)?
- Does my code use Allocatable Arrays (need F2PY again)?

Additional info

- Your Fortran files and libraries need to compiled by the same compiler that you specify for the python-fortran software to use.

Connecting C to Python

- It is quite easy to add new built-in modules to Python, if you know C.
- *Python extension modules* can do two things that can't be done directly in Python, they can:
  - implement new built-in object types
  - call C library functions and system calls.
- To support extensions, the Python API (Application Programmers Interface) defines a set of functions, macros and variables that provide access to most aspects of the Python run-time system.
- The Python API is incorporated in a C source file by including the header "**Python.h**".
- The compilation of an extension module depends on its intended use as well as on your system set-up details are given in later chapters.

The Python API in C: A simple example (1)

- Let's create an extension module called "spam" and let's say we want to create a Python interface to the C library function system(). This function takes a null-terminated character string as argument and returns an integer. We want this function to be callable from Python as follows:

```
>>> import spam
>>> status = spam.system("ls -l")
```

- Begin by creating a file spammodule.c.

  (Historically, if a module is called "spam", the C file containing its implementation is called spammodule.c; if the module name is very long, like "spammify", the module name can be just spammify.c.)

*Much of the information in this document was stolen from the official python documentation at: http://www.python.org

The Python API in C: A simple example (2)

- The first line of our file can be:

```
#include <Python.h>
```

- which pulls in the Python API (you can add a comment describing the purpose of the module and a copyright notice if you like). Since Python may define some pre-processor definitions which affect the standard headers on some systems, you must include **Python.h** before any standard headers are included.

*Much of the information in this document was stolen from the official python documentation at: http://www.python.org

The Python API in C: A simple example (3)

- All user-visible symbols defined by Python.h have a prefix of "Py" or "PY", except those defined in standard header files.
- For convenience, and since they are used extensively by the Python interpreter, "Python.h" includes a few standard header files: **<stdio.h>, <string.h>, <errno.h>,** and **<stdlib.h>.** If the latter header file does not exist on your system, it declares the functions malloc(), free() and realloc() directly.

*Much of the information in this document was stolen from the official python documentation at: http://www.python.org

The Python API in C: A simple example (4)

- The next thing we add to our module file is the C function that will be called when the Python expression “spam.system(*string*)“ is evaluated (we'll see shortly how it ends up being called):

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
char *command;
int sts;
if (!PyArg_ParseTuple(args, "s", &command))
   return NULL;
sts = system(command);
return Py_BuildValue("i", sts);
}
```

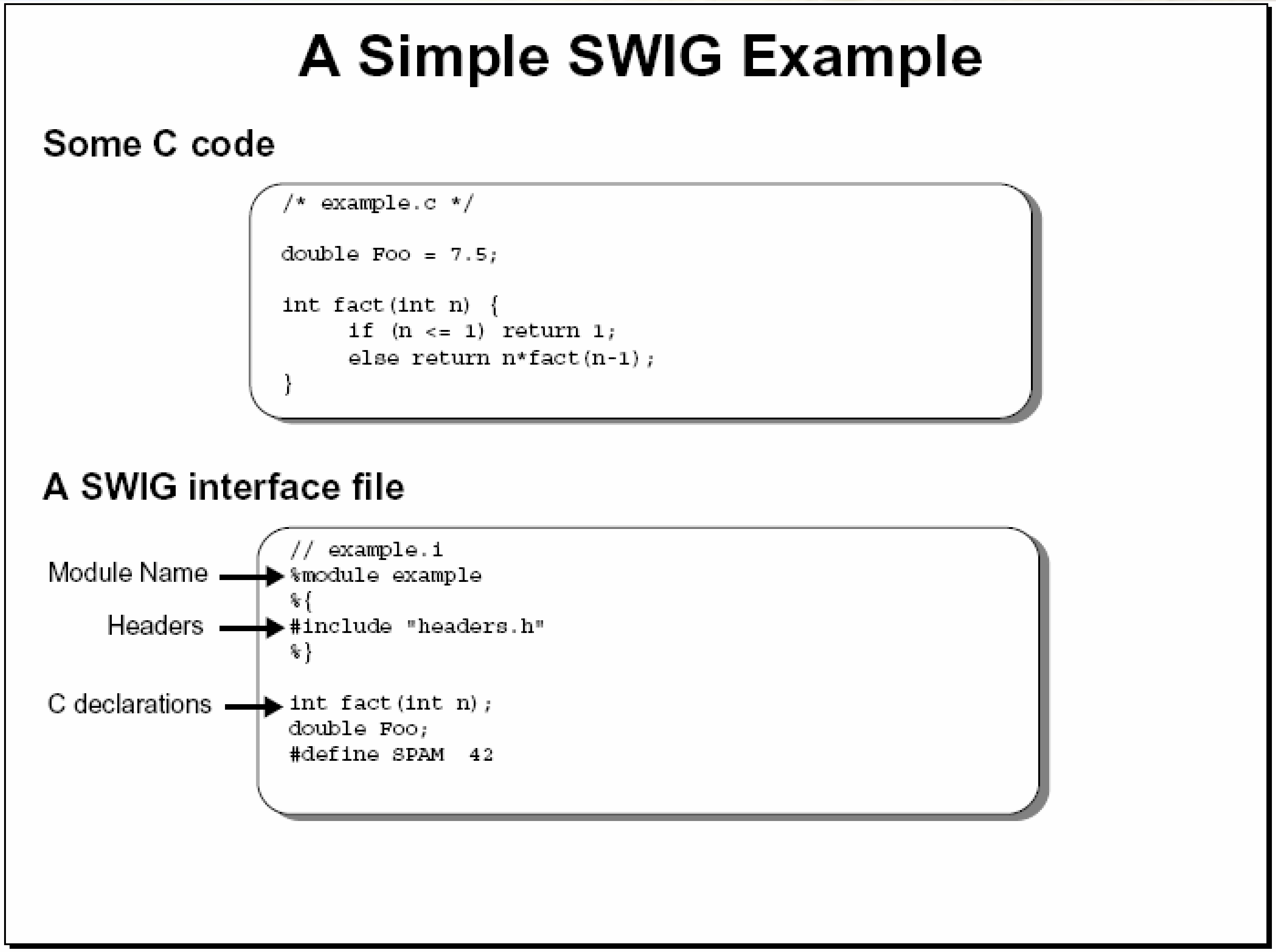
*Much of the information in this document was stolen from the official python documentation at: http://www.python.org

SWIG (Simplified Wrapper and Interface Generator)

- SWIG is a useful tool that allows you to create python wrappers for C code with very little knowledge of the Python C API (but it might not always work).
- It works by taking the declarations found in C/C++ header files and using them to generate the wrapper code that scripting languages need to access the underlying C/C++ code.
- The SWIG interface compiler also connects programmes written in C and C++ with other languages including Perl, Ruby, and Tcl.

*Much of the information in this document was stolen from the official python documentation at: http://www.swig.org/papers/PyTutorial98/PyTutorial98.pdf

SWIG Example (1)



A Simple SWIG Example

Some C code

```
/* example.c */

double Foo = 7.5;

int fact(int n) {
      if (n <= 1) return 1;
      else return n*fact(n-1);
}
```

A SWIG interface file

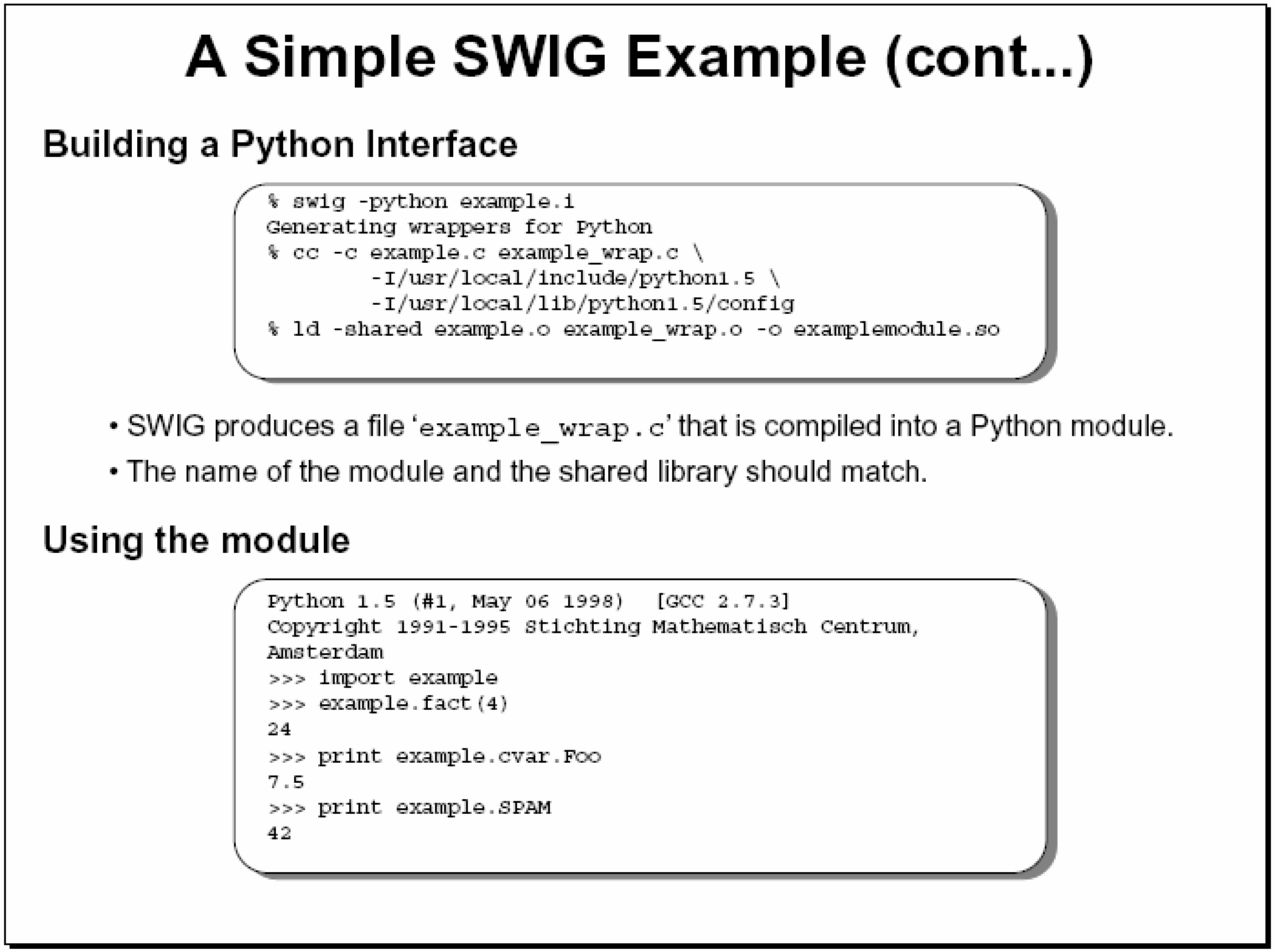
Module Name
Headers
C declarations

```
// example.i
%module example
%{
#include "headers.h"
%}

int fact(int n);
double Foo;
#define SPAM  42
```

*Much of the information in this document was stolen from the official python documentation at: http://www.swig.org/papers/PyTutorial98/PyTutorial98.pdf

SWIG Example (2)



A Simple SWIG Example (cont...)

Building a Python Interface

```
% swig -python example.i
Generating wrappers for Python
% cc -c example.c example_wrap.c \
        -I/usr/local/include/python1.5 \
        -I/usr/local/lib/python1.5/config
% ld -shared example.o example_wrap.o -o examplemodule.so
```

- SWIG produces a file '`example_wrap.c`' that is compiled into a Python module.
- The name of the module and the shared library should match.

Using the module

```
Python 1.5 (#1, May 06 1998)  [GCC 2.7.3]
Copyright 1991-1995 Stichting Mathematisch Centrum,
Amsterdam
>>> import example
>>> example.fact(4)
24
>>> print example.cvar.Foo
7.5
>>> print example.SPAM
42
```

*Much of the information in this document was stolen from the official python documentation at: http://www.swig.org/papers/PyTutorial98/PyTutorial98.pdf